Systèmes d'Exploitation - ENSIN6U3

Outils de synchronisation

Leonardo Brenner [1] Jean-Luc Massat [2]

[1] Leonardo.Brenner@univ-amu.fr

[2] Jean-Luc.Massat@univ-amu.fr

Aix-Marseille Université

Faculté des Sciences

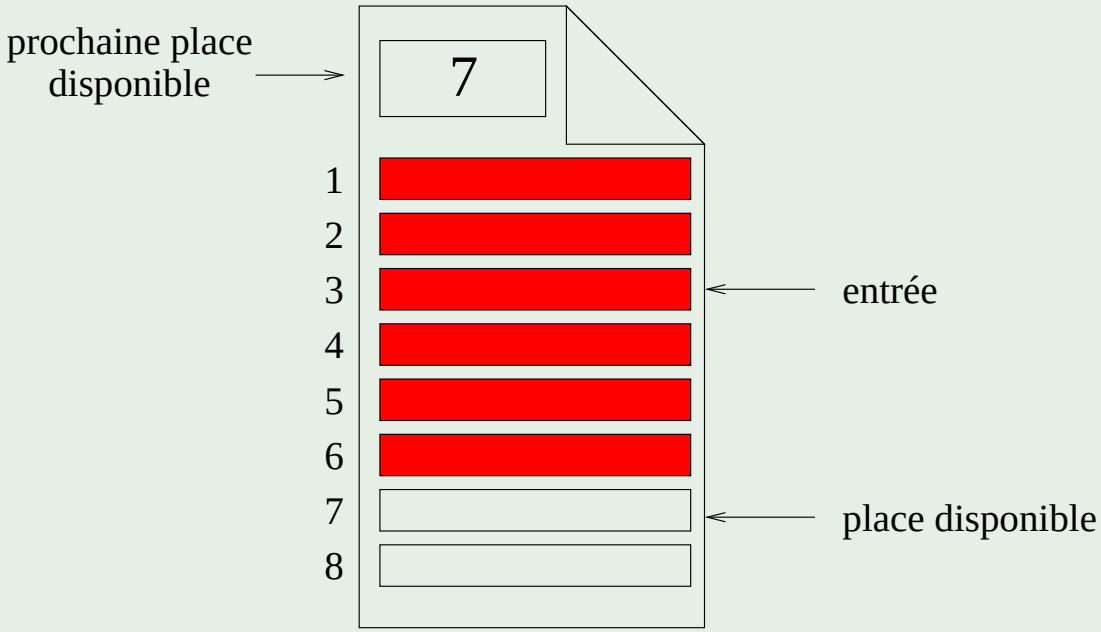
Table de matière

Table de matière

Fichier partagé (1/2)

Cas d'un annuaire partagé

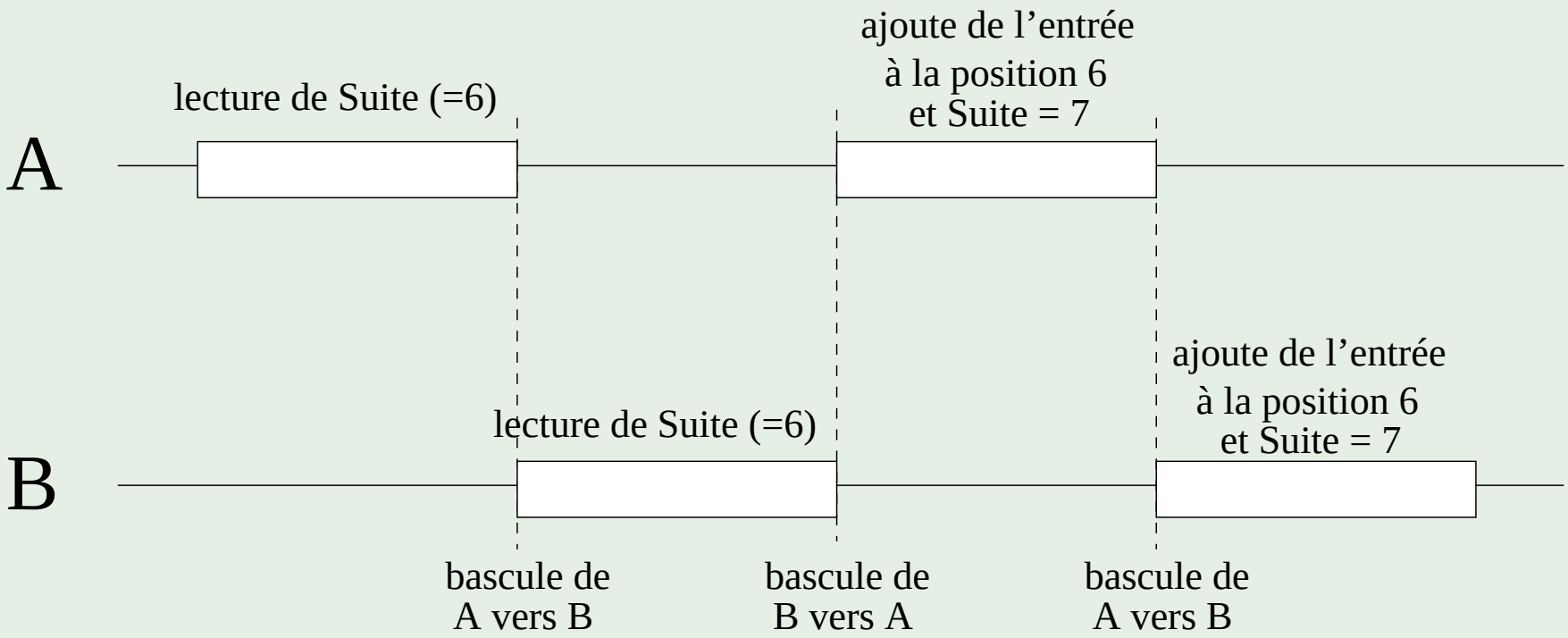
- Supposons un annuaire stocké dans un fichier contenant plusieurs entrées de taille fixe :
  $\Longrightarrow$ L'annuaire est accessible par plusieurs processus.
- Chaque entrée est stockée à une position donnée dans le fichier ;
- En tête de fichier, le numéro de la prochaine place disponible est indiqué (nous l'appellerons Suite).



Fichier partagé (2/2)

Scénario possible avec l'annuaire partagé

$\Longrightarrow$ Que se passe-t-il si deux processus A et B désirent ajouter simultanément une nouvelle entrée à l'annuaire ?



Accès à des ressources partagées

Pile partagée

Soit une pile partagée par plusieurs threads :

```
pile = structure
   | sommet : entier ;
   | data : tableau [ 1 .. Max ] de entier ;

procédure empiler( var p : pile ; d : entier )
   | p.sommet := p.sommet + 1 ;
   | p.data[ p.sommet ] := d ;
```

Possibilité d'exécution

Il est possible que nous ayons

```
thread1      :     empiler( p, 10 )
thread1      :     p.sommet := p.sommet + 1 ;
                — — interruption — —
thread2      :     empiler( p, 20 )
thread2      :     p.sommet := p.sommet + 1 ;
thread2      :     p.data[ p.sommet ] := 20 ;

   :                                :    :
   :                                :    :
        — — retour au thread 1 — —
thread1      :     p.data[ p.sommet ] := 10 ;
```

Sections critiques

Ressources critiques

Les ressources logicielles ou matérielles qui posent des problèmes sont dites **critiques**.

Section critiques

Les portions de code qui manipulent ces ressources critiques sont appelées des **sections critiques**. Ces sections doivent être exécutées en **exclusion mutuelle**

Utilité

Les notions de **ressource critique** et **section critique** sont utiles :

- pour les threads d'un processus utilisateur ;
- pour les données du système d'exploitation partagées par les processus.

Le problème de l'exclusion mutuelle

Forme des programmes

⟨initialisation⟩ exécuté une seule fois

⋮

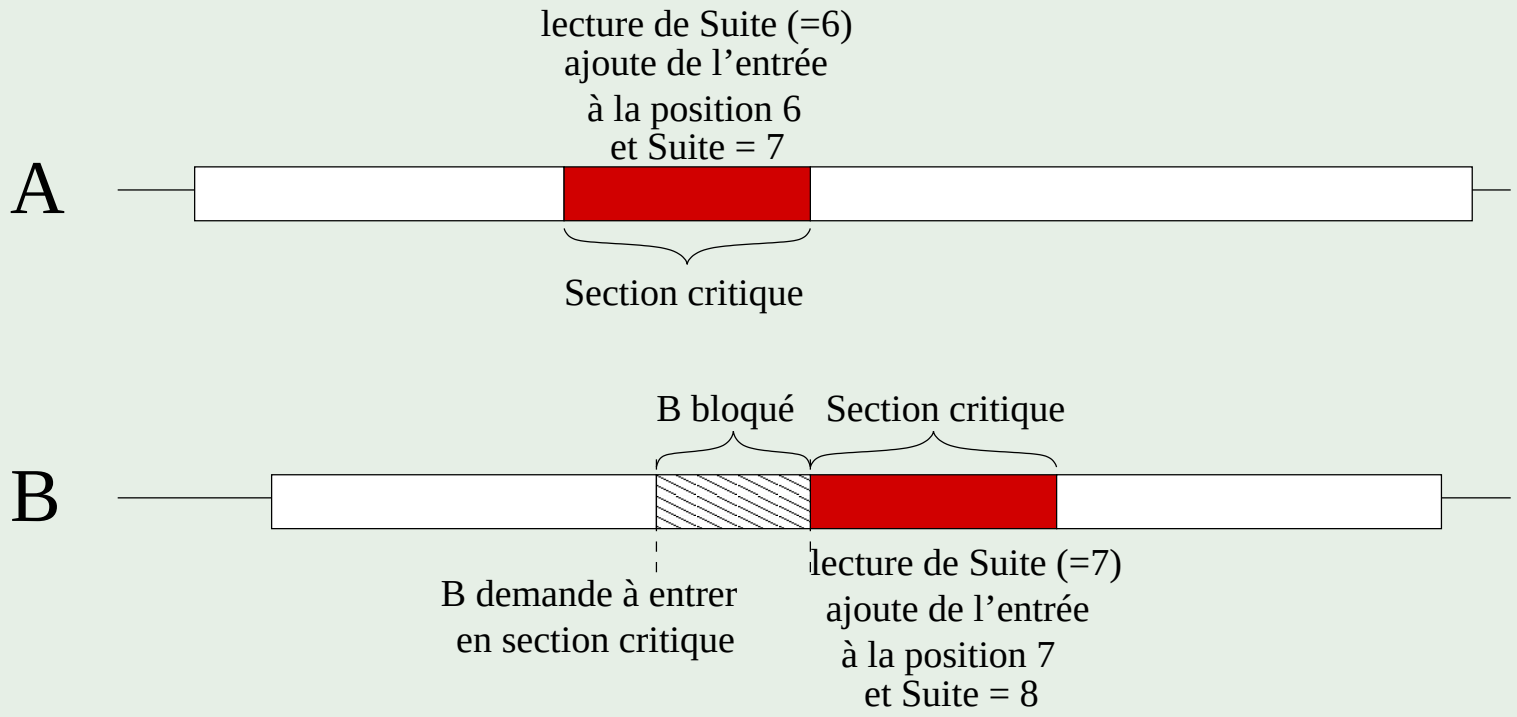
⟨prologue⟩
⟨**section critique**⟩
⟨épilogue⟩

Contraintes

- Il existe **au plus** un processus en section critique (S.C.) ;
- Les processus ne sont pas bloqués sans raison (absence de **privation**) ;
- Les sections ⟨prologue⟩ et ⟨épilogue⟩ sont les mêmes pour tous les processus (**uniformité**) ;
- Le blocage du processus en S.C. ne doit pas entraîner de privation (**tolérance aux pannes**).

Section critique

Scénario avec les sections critiques



Section critique

Pile partagée - nouvelle formulation

La procédure empiler devient :

```
procédure empiler( var p : pile ; d : entier )
    ⟨prologue⟩
        p.sommet := p.sommet + 1 ;
        p.data[ p.sommet ] := d ;
    ⟨épilogue⟩
```

Section critique

Pile partagée - nouvelle exécution

thread1 : empiler(p, 10)
thread1 : ⟨prologue⟩
thread1 : p.sommet := p.sommet + 1 ;
— — **interruption** — —
thread2 : empiler(p, 20)
thread2 : ⟨prologue⟩
— — blocage du thread 2 — —
⋮ ⋮ ⋮
— — retour au thread 1 — —
thread1 : p.data[p.sommet] := 10 ;
thread1 : ⟨épilogue⟩
⋮ ⋮ ⋮
— — reprise du thread 2 — —
thread2 : p.sommet := p.sommet + 1 ;
thread2 : p.data[p.sommet] := 20 ;
thread2 : ⟨épilogue⟩

Table de matière

Attente active - Principe

Description

- Solution logicielle au problème de l'exclusion mutuelle
- Une variable unique du type booléene est **partagée** entre tous les processus
- Lorsqu'un processus veut entrer en section critique :
  1. Si la variable est à faux, le processus attend qu'elle passe à vrai ;
  2. Si la variable est à vrai, le processus la place à faux et entre en section critique ;
  3. Lorsque la section critique est finie, la variable est replacée à vrai.

Une solution d'attente active

Description

Soit *libre* une variable **partagée** de type booléenne pour coder l'exclusion mutuelle :

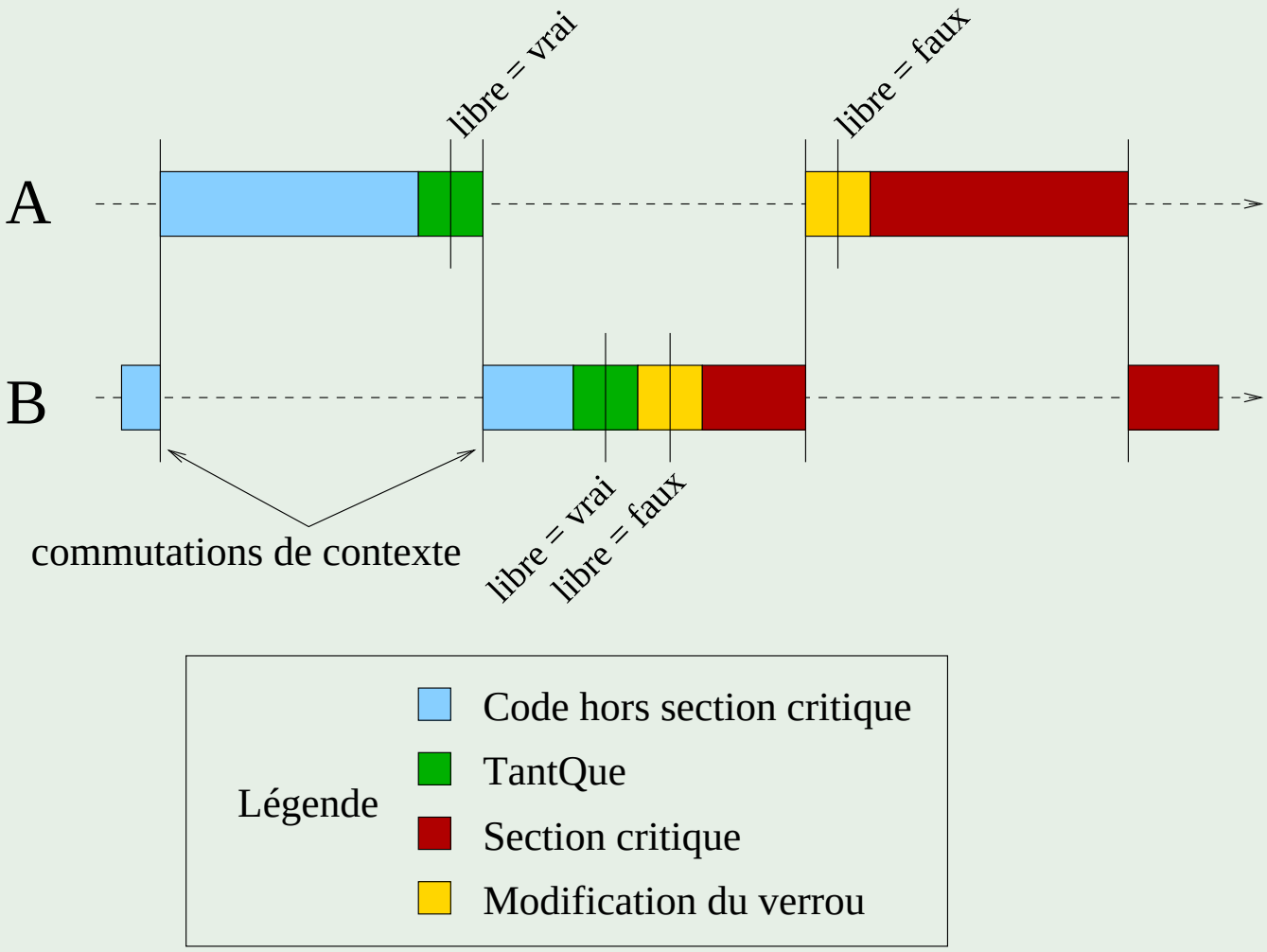
⟨init⟩	(1)	libre := vrai ;
		⋮
⟨prologue⟩	(2)	**si** (libre = faux) **aller en** (2)
	(3)	libre := faux
	(4)	⟨section critique⟩
⟨épilogue⟩	(5)	libre := vrai

Critiques

- Consommation de temps processeur ;
- L'exclusion mutuelle n'est pas toujours respectée. Le test et la modification de la variable ne sont pas atomique.

Problème de cette solution

Exemple d'une double entrée en exclusion mutuelle



Blocage des interruptions (1/3)

Rappel : interruption matérielle

Une interruption matérielle est une action déclenchée au niveau du matériel et non au niveau logiciel.

Exemple de code

```
⟨init⟩          (1)   libre := vrai ;

⟨prologue⟩      (2)   soit p une variable privée
                (3)   répéter
                (4)       ⟨masquer les interruptions⟩
                (5)       p := libre ;
                (6)       libre := faux ;
                (7)       ⟨rétablir les interruptions⟩
                (8)   jusqu'à (p = vrai)

                          ⋮
                      ⟨section critique⟩
                          ⋮
⟨épilogue⟩      (9)   libre := vrai
```

Blocage des interruptions (2/3)

Exemple d'écriture sur un disque dur

- Le pilote informe le contrôleur de périphérique (CP) des actions à réaliser (exemple : écriture d'octets) ;
- Le CP démarre le périphérique et envoie les actions ;
- Lorsque les actions sont terminées, CP signale au contrôleur d'interruptions (CI) ;
- Si le CI est prêt, il en informe le CPU ;
- Le CI place le numéro du périphérique dans le bus pour avertir le CPU ;
- Le CPU décide quand il prend en charge l'interruption.

Blocage des interruptions (3/3)

Entrée en section critique

- Lorsqu'un processus entre en section critique, il désactive les interruptions matérielles ;
- Dans ce cas, l'horloge ne peut pas envoyer d'interruption :
  $\Longrightarrow$ Le CPU ne peut plus basculer d'un processus à un autre.

Problèmes

- L'ordre d'arrivée n'est pas respectée ;
- Il faut donner la possibilité aux processus utilisateur de contrôler les interruptions (dangereux) ;
- Dans le cas d'un système multi-processeurs, la désactivation des interruptions n'affecte qu'un seul processeur ;
- Que se passe-t-il si le processus ne rend pas la main après la désactivation ?
  $\Longrightarrow$ Risque de privation.

Alternance stricte (1/2)

Description

- Les deux processus n'entrent pas en section critique simultanément ;
- Ils y rentrent à tour de rôle.

Problème

- Un processus ne peut pas entrer deux fois d'affilé dans la section critique ;
- L'ordre est fixé à priori avec alternance ;
- Risque de privation (blocage inutile) ;
- Il n'y a pas de mémoire de l'état des processus.

Alternance stricte (2/2)

Exemple de code

Soit *tour* une variable entière partagée :

var tour : **entier** ;

Voila le codage pour le processus P_i :

⟨initialisation⟩	(1)	tour := 0 ;
⟨prologue⟩	(2)	**répéter**
	(3)	**jusqu'à** (tour = i)
⟨épilogue⟩	(4)	tour := (1 − i) ;

Solution de Peterson pour deux processus

Description

- Chaque processus est identifié par un numéro (ici 0 ou 1) ;
- Données partagées par tous les processus :
  - Variable tour ;
  - Tableau états : indique si le processus désire entrer en section critique.

Exemple de code

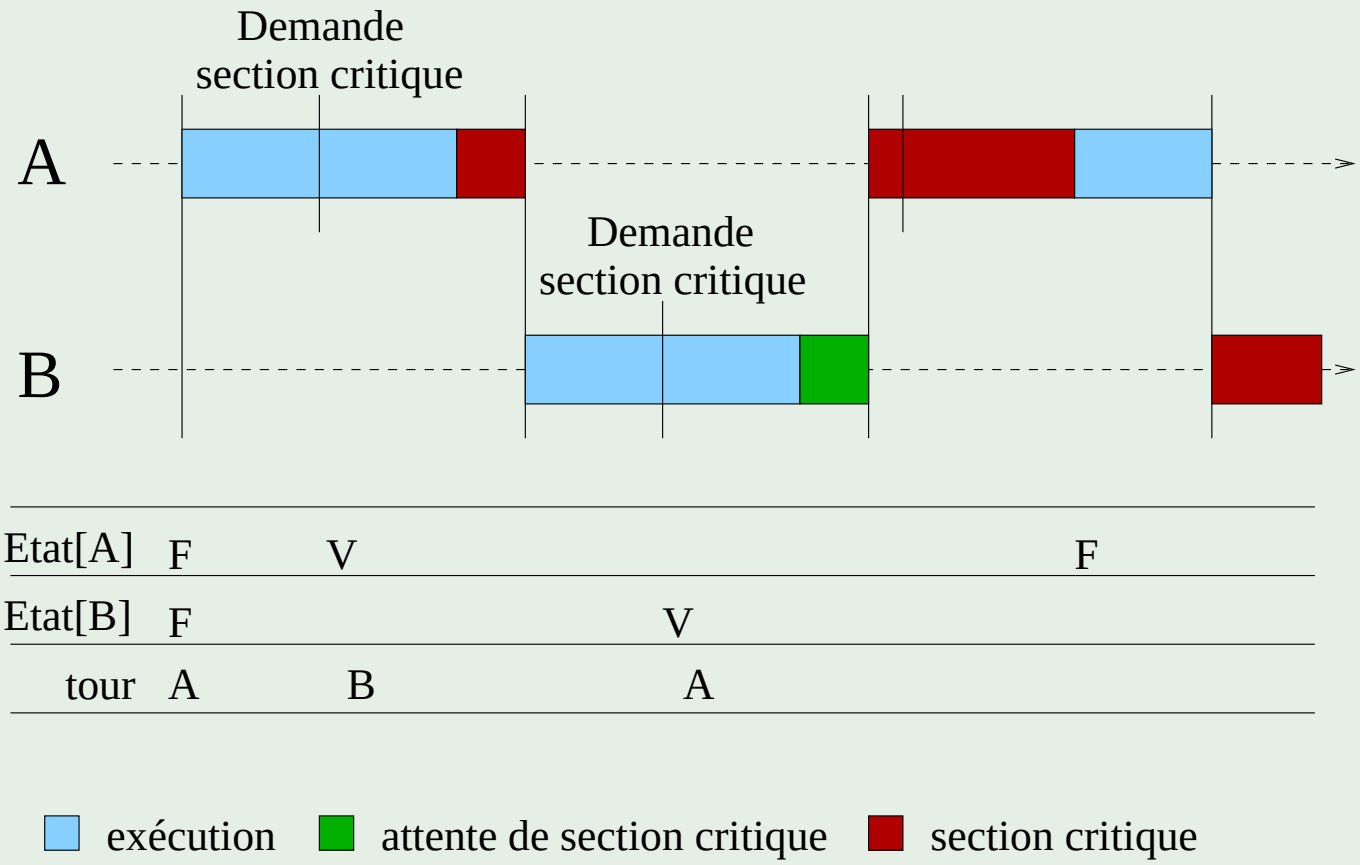
var tour : **entier** ;
états : **tableau** [0 .. 1] de **booléen** ;

Voila le codage pour le processus P_i :

⟨initialisation⟩	(1)	tour := 0 ;
	(2)	états := (faux, faux) ;
⟨prologue⟩	(3)	états[i] := vrai ;
	(4)	tour := $1 - i$;
	(5)	**répéter**
	(6)	**jusqu'à** (tour = i) **ou** (états[$1 - i$] = faux)
⟨épilogue⟩	(7)	états[i] := faux ;

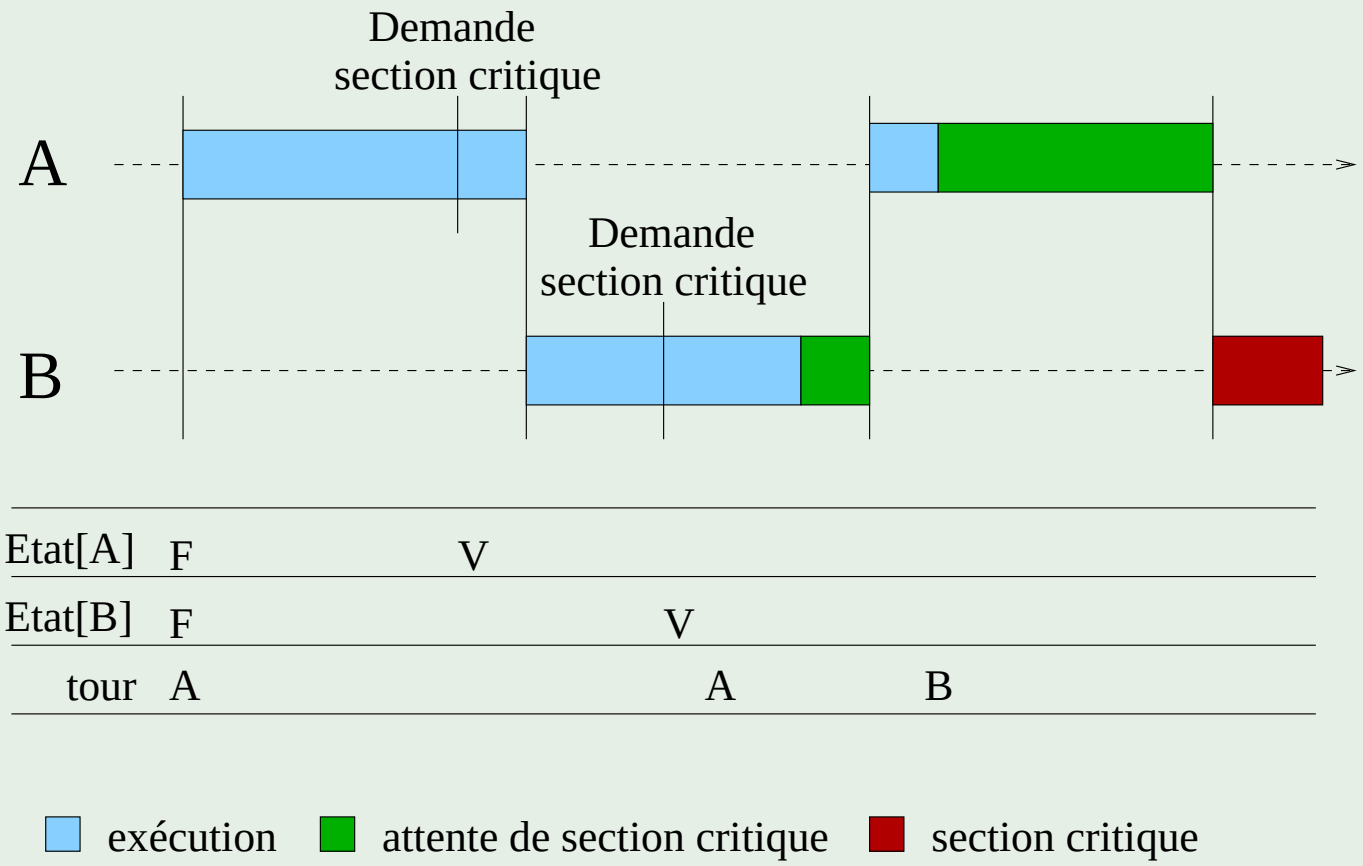
Exemple de l'algorithme de Peterson (1/2)

Exemple d'exécution



Exemple de l'algorithme de Peterson(2/2)

Exemple d'exécution



Solution matérielle (1/2)

Description

On introduit une nouvelle instruction *Test and Set* pour

- garantir l'atomicité d'une modification,
- construire une solution basée sur l'attente active valable en multi-processeurs :

Définition de *Test and Set*

instruction TAS(**var** m : **entier** , **var** verrou : **entier**)
| ⟨bloquer la case mémoire verrou⟩
| | m := verrou
| | verrou := 0
| ⟨débloquer la case mémoire verrou⟩
| CO := CO + ⟨taille de l'instruction TAS⟩

Solution matérielle (2/2)

Utilisation de Test and Set

Codage de l'exclusion mutuelle avec une variable partagée *mutex* et une variable privée *p*.

```
⟨initialisation⟩   (1)   mutex := 1 ;
⟨prologue⟩         (2)   répéter
                   (3)     | TAS(p, mutex) ;
                   (4)   jusqu'à (p = 1)
                   (5)   ⟨section critique⟩
⟨épilogue⟩         (6)   mutex := 1 ;
```

Critiques

- Consommation de temps processeur.
- Risque de privation (cela peut s'arranger).
- Le processeur doit garantir l'exclusion mutuelle.

C'est une solution utilisable uniquement sur des **séquences brèves**.

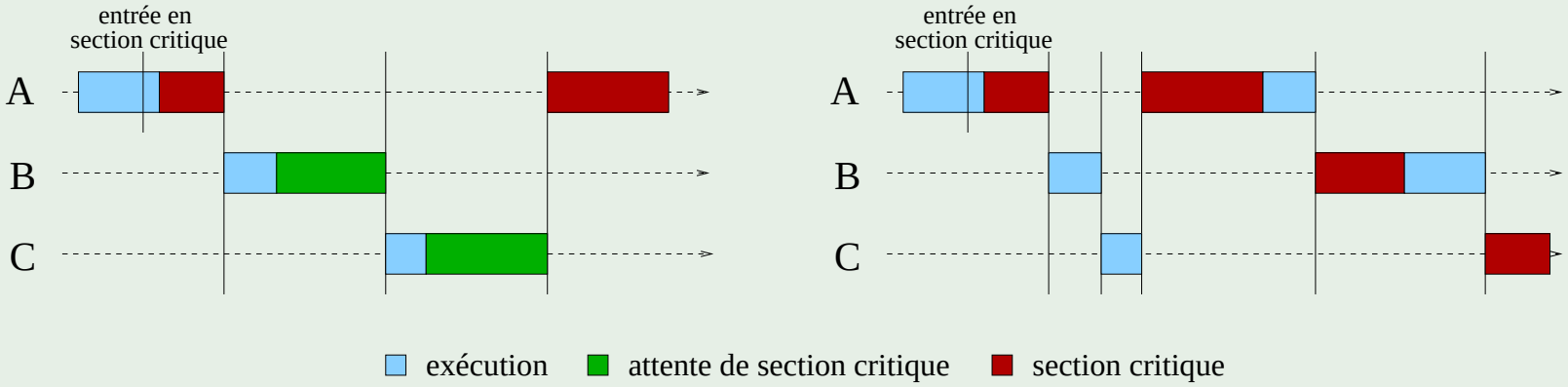
Table de matière

Attente active vs attente passive

Définitions

- *Attente active* : le processus est en attente de section critique mais consomme du CPU ;
- *Attente passive* : le processus est en attente de section critique mais ne consomme pas de CPU.

Exemples



Les verrous (1/4)

Objectifs

- ne plus perdre de temps CPU,
- simplicité de la solution.

Définition des verrous

Un **verrou** est une structure de donnée **partagée** du système d'exploitation.

verrou : **structure**
  libre : booléen ;
  f : file FIFO de processus ;

procédure init(**var** *v* : verrou)
  v.libre := **vrai** ;
  v.f := {} ;

Les verrous (2/4)

Exemple de code

Pour un verrou donné, les deux procédures ci-dessous s'exécutent en exclusion mutuelle.

```
procédure prendre(var v : verrou)
    si (v.libre = faux ) alors
        ⟨soit P le processus appelant⟩
        ⟨entrer P dans la file v.f⟩
        ⟨suspendre le processus P⟩
    sinon
        v.libre := faux ;
    fin si

procédure libérer(var v : verrou)
    si ⟨la file v.f est vide⟩ alors
        v.libre := vrai ;
    sinon
        v.libre := faux ;
        ⟨sortir un processus Q de la file v.f⟩
        ⟨réveiller le processus Q⟩
    fin si
```

Les verrous (3/4)

L'exclusion mutuelle avec les verrous

Soit

var mutex : verrou ;

Le code de l'exclusion mutuelle s'écrit

⟨initialisation⟩ (1) init(*mutex*) ;

⟨prologue⟩ (2) prendre(*mutex*) ;

⋮

⟨section critique⟩

⋮

⟨épilogue⟩ (3) libérer(*mutex*) ;

Les verrous (4/4)

Difficultés des verrous

Soit deux processus qui partagent deux ressources :

```
var   mutex1 : verrou ;
      mutex2 : verrou ;

      init(mutex1) ;
      init(mutex2) ;
```

$\langle P_1 \rangle$		$\langle P_2 \rangle$	
(1)	prendre(*mutex1*) ;	(1')	prendre(*mutex2*) ;
(2)	| prendre(*mutex2*) ;	(2')	| prendre(*mutex1*) ;
(3)	| | ⟨section critique⟩	(3')	| | ⟨section critique⟩
(4)	| libérer(*mutex2*) ;	(4')	| libérer(*mutex1*) ;
(5)	libérer(*mutex1*) ;	(5')	libérer(*mutex2*) ;

Il y a blocage pour la séquence **1 ... 1' ... 2 ... 2'**

Les sémaphores (1/3)

Motivations

- Un verrou ne sait pas compteur... donc, il faut remplacer le drapeau par un compteur ;
- En 1965, Dijkstra propose de nouvelles variables nommées **sémaphores** ;

Définition

Un **sémaphore** est une structure de donnée **partagée** du système d'exploitation.

sémaphore : **structure**
  c : entier ;
  f : file FIFO de processus ;

procédure init(**var** s : sémaphore ; c_0 : entier)
  vérifier que $c_0 \geq 0$;
  $s.c := c_0$;
  $s.f := \{\}$;

Les sémaphores (2/3)

Agir sur les sémaphores

Pour un sémaphore donné, les deux procédures ci-dessous s'exécutent en exclusion mutuelle.

procédure P(**var** s : sémaphore)
- $s.c := s.c - 1$
- **si** $(s.c < 0)$ **alors**
  - ⟨soit P le processus appelant⟩
  - ⟨entrer le processus P dans la file $s.f$⟩
  - ⟨suspendre le processus P⟩
- **fin si**

procédure V(**var** s : sémaphore)
- $s.c := s.c + 1$
- **si** $(s.c \leq 0)$ **alors**
  - ⟨sortir un processus Q de la file $s.f$⟩
  - ⟨reprendre le processus Q⟩
- **fin si**

P pour *proberen* (tester) ou *wait*
V pour *verhogen* (incrémenter) ou *signal*

Les sémaphores (3/3)

Utilisations des sémaphores

Soient trois processus qui exploitent la même ressource :

	P_1	P_2	P_3
(1)	P(s)		
(2)		P(s)	
(3)			P(s)
(4)	V(s)		
(5)		V(s)	
(6)			V(s)

La trace de l'exécution donne :

	action	(2,{})	P_1	P_2	P_3
(1)	P(s)	(1,{})	SC	A	A
(2)	P(s)	(0,{})	SC	SC	A
(3)	P(s)	(-1,{P_3})	SC	SC	S
(4)	V(s)	(0,{})	A	SC	SC
(5)	V(s)	(1,{})	A	A	SC
(6)	V(s)	(2,{})	A	A	A

Sémaphores simulés par des verrous (1/2)

Définition

Un sémaphore peut être défini par la structure suivante :

sémaphore : **structure**
- *blocage* : verrou ;
- *mutex* : verrou ;
- *c* : entier ;

procédure init(**var** *s* : sémaphore ; c_0 : entier)
- vérifier que $c_0 \geq 0$;
- $s.c := c_0$
- init(s.blocage) ;
- init(s.mutex) ;

Sémaphores simulés par des verrous (2/2)

Procedure P

procédure P(**var** s : sémaphore)
- prendre(s.mutex) ;
- $s.c := s.c - 1$
- **si** $(s.c = -1)$ **alors**
  - libérer(s.mutex) ;
  - prendre(s.blocage) ;
  - prendre(s.blocage) ;
- **sinon si** $(s.c < 0)$ **alors**
  - libérer(s.mutex) ;
  - prendre(s.blocage) ;
- **sinon**
  - libérer(s.mutex) ;
- **fin si**

Procedure V

procédure V(**var** s : sémaphore)
- prendre(s.mutex) ;
  - $s.c := s.c + 1$
  - libérer(s.blocage) ;
- libérer(s.mutex) ;

Sémaphores privés

Définition

Un sémaphore s est un sémaphore *privé* d'un processus si seul ce processus peut exécuter $P(s)$. Les autres processus ne pouvant agir que par $V(s)$.

Utilisation

Soit *sempriv* un sémaphore privé de P_1 et *mutex* un sémaphore ordinaire :

P_1 :
P(*mutex*)
si ⟨le blocage est inutile⟩ **alors**
  V(*sempriv*)
fin si
V(*mutex*)
P(*sempriv*)

P_2 :
P(*mutex*)
si ⟨le processus P_1 doit être débloqué⟩ **alors**
  V(*sempriv*)
fin si
V(*mutex*)

Problème du producteur/consommateur

Définition

Il existe un tampon de taille **limité** entre le producteur et le consommateur.

Algorithme du producteur

répéter
  ⟨produire un message⟩
  ⟨le déposer dans le tampon⟩
jusqu'à ...

Algorithme du consommateur

répéter
  ⟨prélever un message depuis le tampon⟩
  ⟨le consommer⟩
jusqu'à ...

Codage producteur/consommateur (1/2)

Initialisation

NPlein : sémaphore = $(0, \{\})$

Producteur

répéter
⟨produire un message⟩
⟨le déposer dans le tampon⟩
V(NPlein) ;
jusqu'à ...

Consommateur

Le consommateur consomme si le tampon n'est pas vide :

répéter
P(NPlein) ;
⟨consommer⟩
jusqu'à ...

Codage producteur/consommateur (2/2)

Initialisation

NPlein : sémaphore = $(0, \{\})$
NVide : sémaphore = $(n, \{\})$

Producteur

Le producteur produit si le tampon n'est pas plein :

répéter
  P(NVide) ;
  ⟨produire un message⟩
  ⟨le déposer dans le tampon⟩
  V(NPlein) ;
jusqu'à ...

Consommateur

Le consommateur consomme si le tampon n'est pas vide :

répéter
  P(NPlein) ;
  ⟨consommer⟩
  V(NVide) ;
jusqu'à ...

Sémaphores à messages (1/2)

Définition

Un sémaphore à message est une version modifiée des sémaphore classique qui permet de transmettre un message entre les processus.

séma-mesg : **structure**
- *c* : entier ;
- demandeurs : file FIFO de processus ;
- messages : file FIFO de messages ;

Sémaphores à messages (2/2)

Procédure *Pm*

procédure Pm(**var** s : séma-mesg ; **var** m : données)
  $s.c := s.c - 1$
  si $(s.c < 0)$ **alors**
    ⟨soit P le processus appelant⟩
    ⟨entrer le processus P dans la file s.demandeurs⟩
    ⟨suspendre le processus P⟩
  fin si
  ⟨sortir m de la file s.messages⟩

Procédure *Vm*

procédure Vm(**var** s : séma-mesg ; m : données)
  ⟨entrer m dans la file s.messages⟩
  $s.c := s.c + 1$
  si $(s.c \leq 0)$ **alors**
    ⟨sortir un processus Q de la file s.demandeurs⟩
    ⟨reprendre le processus Q⟩
  fin si

Régions critiques (1/2)

Principe

- Une région critique est un autre outil de synchronisation de haut niveau.
- On peut accéder à la variable *v* uniquement dans une instruction **region** de la forme suivante :

  région *v* **quand** B **faire** S

- B est une expression logique
- S est le code critique à exécuter

Régions critiques (2/2)

Définition

```
pile = structure partagée
    | sommet : entier ;
    | data : tableau [ 1 .. Max ] de entier ;
```

Procédure empiler

```
procédure empiler( var p : pile ; d : entier )
    | région p quand (sommet < Max)
    |     | sommet := sommet + 1 ;
    |     | data[ sommet ] := d ;
    | fin de région
```

Procédure dépiler

```
procédure dépiler( var p : pile ; var d : entier )
    | région p quand (sommet > 0)
    |     | d := data[ sommet ] ;
    |     | sommet := sommet - 1 ;
    | fin de région
```

Régions critiques simulées par sémaphores (1/2)

Définitions

L'instruction

région p **quand** (*condition*)
  | ⟨*région critique*⟩
fin de région

peut se coder par

```
var   mutex        : sémaphore = (1, {}) ;
      blocage      : sémaphore = (0, {}) ;
      nbEnAttente  : entier = (0) ;
```

Régions critiques simulées par sémaphores (2/2)

Définitions -continuation

P(mutex) ;
tant que non (*condition*)
    nbEnAttente := nbEnAttente + 1 ;
    V(mutex) ;
    P(blocage) ;
    P(mutex) ;

⟨*région critique*⟩

répéter nbEnAttente **fois** V(blocage) ;
nbEnAttente := 0 ;
V(mutex) ;